

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ В.Н. КАРАЗІНА
ФАКУЛЬТЕТ КОМП'ЮТЕРНИХ НАУК

КУРСОВА РОБОТА
з дисципліни «Теорія алгоритмів»
Тема «2-3 дерево»

Виконав студент 2 курсу, групи КС-21
Синюк Валентин Миколайович
Керівник: доц. Щебенюк В. С.

Харків – 2018

ЗМІСТ

Вступ.....	3
1 Теоретичні основи	
1.1 Теоретичні відомості та властивості	4
1.2 Область застосування прикладного рішення	4
1.3 Переваги та недоліки алгоритму	5
1.4 Основні операції для 2-3 дерева	5
2 Методи та модель реалізації прикладної задачі	
2.1 Інструментальні засоби реалізації прикладної задачі	8
2.2 Модель програмного продукту.....	10
3 Реалізація прикладного рішення	
3.1 Побудова програмного продукту	11
3.2 Тестування програмного продукту.....	14
Висновок.....	17
Список джерел інформації.....	18
Додатки.....	19

ВСТУП

В роботі буде описано процес розробки програмної моделі (ПМ), яка представляє собою таку структури даних, як 2-3 дерево. ПМ буде працювати, за певним сценарієм, тобто конфігурації вхідних даних (параметрів) та методів (дій), які будуть обробляти їх.

Буде вивчена відповідна література по проблематиці реалізації такої структури даних, зокрема будуть розглянуті ключові поняття та методики по роботі з нею. Розуміння методів, дадуть змогу практично застосувати набуті знання.

Структура курсової роботи складається зі вступу, трьох розділів, висновку, списку джерел інформації та додатків.

В першому розділі, будуть надані теоретичні відомості та область застосування прикладного рішення. Також, будуть розглянуті переваги та недоліки алгоритму. Окрім цього, будуть якісно описані основні методи, для роботи з 2-3 деревом, які представляють основний функціонал. Даний розділ має теоретичну цінність.

У другому розділі, будуть детально розглянуті методи та модель прикладної задачі. Також, будуть описані різноманітні інструментальні засоби, для реалізації та візуалізації програмного продукту.

В третьому розділі, буде детально описуватися реалізація прикладного рішення, тобто, будуть розглянуті основні положення програмного продукту та відповідно буде зображено його побудова та тестування. Даний розділ має практичну цінність.

1 ТЕОРЕТИЧНІ ОСНОВИ

1.1 Теоретичні відомості та властивості

2-3 дерево – структура даних, що є В-деревом, кожен вузол якого має або два нащадки і одне поле, або три нащадки і два поля. 2-3 дерево є бінарним деревом пошуку [1]. Це дерево має такі властивості [2]:

- 1). вузол повинен завжди мати перше поле, але необов'язково друге поле;
- 2). якщо обидва поля присутні у вузлі, перше поле завжди менше другого;
- 3). кожен вузол може мати до трьох дочірніх вузлів, але якщо у вузлі є тільки одне поле даних, вузол не може мати більше двох дітей;
- 4). дочірні вузли встановлені так, що дані в першому піддереві менші першого поля даних, дані другого піддерева більші, ніж перше поле даних і менші другого поля даних, а дані третього піддерева більші другого поля даних;
- 5). якщо у вузлі є тільки одне поле даних, використовуйте тільки перший і другий дочірні елементи;
- 6). всі листові вузли з'являються на останньому рівні;
- 7). всі листові вершини знаходяться на одному рівні.

1.2 Область застосування прикладного рішення

Структура 2-3 дерева застосовується для організації індексів у багатьох сучасних системах управління базами даних (СУБД). 2-3 дерево може застосовуватися для структурування (індексування) інформації на жорсткому диску. Час доступу до довільного блоку на диску дуже велике, оскільки воно визначається швидкістю обертання диска і переміщення головки. Тому важливо зменшити кількість вузлів, що переглядаються при кожній операції. Використання пошуку за списком кожен раз для знаходження випадкового блоку могло б привести до надмірно кількості звернень до диска внаслідок необхідності послідовного проходження по всіх його елементах, що передують заданому, тоді як пошук в 2-3 дереві,

завдяки властивостям збалансованості і високим зеленим, дозволяє значно скоротити кількість таких операцій.

1.3 Переваги та недоліки алгоритму

Проста реалізація алгоритмів та існування готових бібліотек для роботи зі структурою 2-3 дерева, забезпечують популярність застосування такої організації пам'яті в найрізноманітніших програмах, що працюють з великими обсягами даних.

У всіх випадках корисне використання простору вторинної пам'яті становить понад 50%. З ростом ступеня корисного використання пам'яті не відбувається зниження якості обслуговування.

В середньому, досить ефективно реалізуються операції вставки та видалення записів; при цьому зберігається природний порядок ключів з метою послідовної обробки, а також відповідний баланс дерева для забезпечення швидкої довільної вибірки. Незмінна впорядкованість по ключу забезпечує можливість пакетної обробки.

Основний недолік 2-3 дерева, полягає у відсутності для них, ефективних засобів вибірки даних (тобто методу обходу дерева), упорядкованих по відмінному від обраного ключа.

1.4 Основні операції для 2-3 дерева

Дамо характеристику основним операціям для роботи з 2-3 деревом.

1 Операція **вставки**. Є 3 варіанти при вставці (додаванню) елемента X в 2-3 дерево [2].

А. Якщо у вузлі немає елементів або він один, то присвоюємо значення X або першому полю, якщо елементів немає, або - в залежності від того, X більше існуючого елемента, чи ні. Якщо X більше, то присвоюємо його другому полю, якщо менше – першому, а існуючий елемент – другому.

Б. Якщо у вузлі є 2 значення, а його батько один, чи батька немає, то потрібно визначити середнє значення, і підняти його в батьківський вузол. Присвоїти значення в перше або друге поле, в залежності від розміру присутнього елемента. Якщо батька немає, то створити його. Далі потрібно створити середнього сина, і привласнити йому одне зі значень крайнього вузла. Якщо ми створюємо нового батька, то елемент із середнім значенням стає батьком, а два інших - лівим і правим нащадками.

В. Якщо у вузлі є обидва елементи, і в його батька так само обидва елементи, то потрібно повторювати дії 2 пункту, поки елементи не займуть відповідне поле.

2 Операція **видалення**. Так само може проходити в декількох варіантах [2]:

А. Якщо в дереві є тільки один вузол, то видаляємо або перше значення, або друге. При видаленні першого значення і наявності другого, присвоюємо друге значення першого поля. Процедура повторюється для листів дерева.

Б. Якщо видається елемент - єдиний в вузлі, то необхідно зробити перестановку. Її зручніше здійснювати з ближнього брата, що має 2 елементи. Береться середньої елемент між батьком і сином з 2 елементами, піднімається в батьківське поле, а два інших – стають його синами. При наявності онуків, необхідно продовжувати сортування до кінця дерева. Якщо у батьків видається елемента є 2 значення, то проводиться та ж сортування, тільки, в залежності від того, є видається елемент лівим, середнім або правим сином, сортується перше чи друге значення батька.

В. Якщо у вузла 2 сина, і один з них - видається елемент, то необхідно опустити значення батька в вузол другого сина, поставити його відповідно значенням (в перше поле, або друге), а батьківський вузол видалити, якщо він є коренем дерева, і вузол спадкоємця зробити новим коренем дерева. В іншому випадку - потрібно використовувати сортування для батьків. Те ж саме

відбувається в разі, якщо батько має 2 значення і 3 сина, один з яких видаляється ключем. Але ми опускаємо значення батька відповідно до того, елемент якого вузла був видалений. Наприклад, при видаленні лівого сина, на його місце опускається елемент лівого поля батька, середній син стає другим елементом лівого сина, другий елемент батька присвоюється першому полю батька, а третій син стає батьком.

3 Операція **пошуку** [2]. Пошук в 2-3 дереві, здійснюється як в бінарному дереві пошуку. Спочатку перевіряємо елементи кореня. Якщо шуканий елемент – значення першого або другого поля – повертаємо його. В іншому випадку, дивимося шуканий елемент, який більше першого значення кореня, але менше другого, або більше другого. В залежності від варіанту – повертаємо значення рекурсивно з лівим вузлом, середнім або правим.

4 Інші, другорядні операції. Ці операції не так важливі як перші три, але без них, не можливо уявити адекватну роботу 2-3 дерева. До таких операцій належать: очищення вмісту дерева, віднаходження мінімального та максимального значень, підрахунок розміру дерева, відображення вмісту дерева, за допомогою трьох різних обходів.

2 МЕТОДИ ТА МОДЕЛЬ РЕАЛІЗАЦІЇ ПРИКЛАДНОЇ ЗАДАЧІ

2.1 Інструментальні засоби реалізації прикладної задачі

Проектування – застосовується на початку розробки програмного продукту. Найчастіше, розробники програмного забезпечення (ПЗ), схилиються до проектування візуальної моделі за принципами каскадної (водоспадної) моделі. Вона представляє собою послідовність розробки ПЗ. Ієрархія такої моделі складається з декількох основних стадій реалізації та впровадження: складання вимог до програмного продукту (ПП), проектування та реалізація програмної моделі (ПМ), верифікація, тестування та технічне обслуговування ПП. Розглянемо етап проектування ПМ.

Інструментальні засоби, які використовуються для проектування, тобто моделювання та візуалізації ПМ, ґрунтуються на UML – уніфікованій мові моделювання [9]. На теперішній день, вже існує досить багато редакторів та засобів для створення UML діаграм. Найпоширенішими програмами є: Microsoft Visio, Visual Paradigm, Eclipse IDE, IntelliJ IDEA.

Для проектування візуальної моделі, яка зображена на рисунку 2.1, використовувався інструмент з середовища розробки програмного забезпечення Eclipse, а саме додатково встановлений модуль (плагін) – «The ObjectAid UML Explorer for Eclipse».

Це легкий та гнучкий інструмент для графічного представлення існуючого коду. За допомогою нього, можна створювати діаграми класів та послідовностей, які автоматично оновлюються не виходячи з синхронізації. Автоматична генерація діаграм з вихідного коду та вбудований функціонал, дозволяють якісно та швидко відтворювати модель прикладної задачі.

Плагін дуже простий у використанні. Управління відбувається за допомогою контекстного меню, при натисканні правої кнопки миші.

Модель ПП буде розроблятися на такій мові програмування, як Java. Дана

мова програмування дає змогу реалізувати основні парадигми об'єктно-орієнтованого програмування (ООП): абстракція, інкапсуляція, успадкування та поліморфізм.

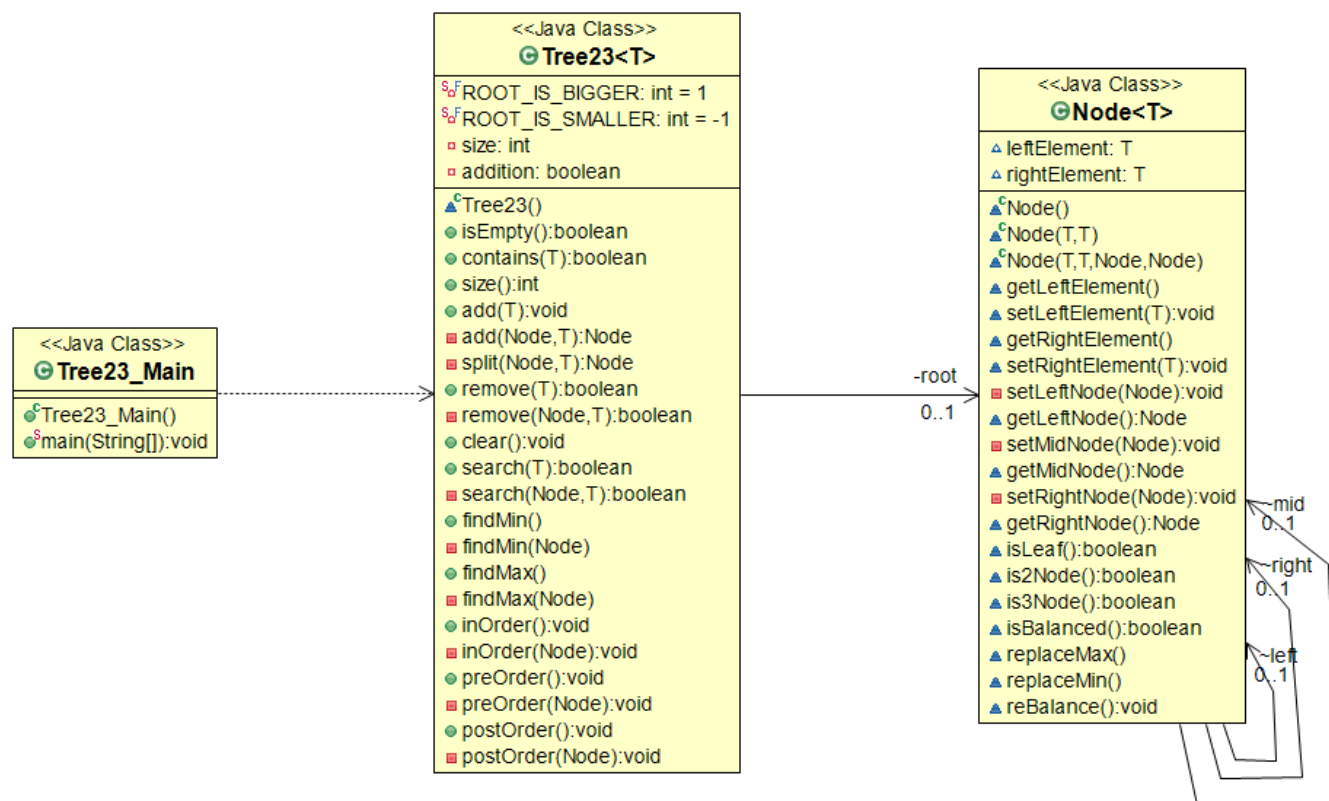


Рисунок 2.1 – UML діаграма рішення ПМ

Виконання програм на Java базується на використанні важливого елементу мови – віртуальної машини (JVM), яка є основою виконуючої системи Java, так званої Java Runtime Environment (JRE). JVM виконує байт-код, попередньо створений з вихідного тексту Java-програми компілятором Java (Javac). Компілятор приймає вихідний код, відповідний специфікації Java Language Specification і повертає байт-код, вже відповідний специфікації Java Virtual Machine Specification.

Другорядним, але не менш важливим елементом програмування на Java є – середовище розробки ПЗ (IDE), яке необхідне для реалізації прикладних задач та

потребує ретельного підбору, вивчення та ознайомлення. Зважаючи на всі переваги та недоліки багатьох широкодоступних IDE, був визначений гідний кандидат, як інструмент, що використовуватиметься – IntelliJ IDEA версії Ultimate. Він представляє собою інтегроване середовище розробки ПЗ для багатьох мов програмування, розроблене компанією JetBrains. Існують дві версії цієї IDE:

- а) Community Edition – безкоштовна версія з урізаним функціоналом;
- б) Ultimate Edition – комерційна повнофункціональна версія.

Порівняємо та надамо стислу характеристику цих версій. Більшість користувачів даної IDE використовують саме безкоштовну версію. Вона підтримує велику кількість бажаних інструментів для розробки ПЗ, наприклад, такі мови програмування як: Java, Groovy, Kotlin, Scala; такі засоби складання як: Gradle, Maven; та такі системи контролю версій як: Git, SVN, Mercurial.

Комерційна версія відрізняється наявністю підтримки багатьох інших важливих структур та технологій JVM, включаючи: Hibernate, Guice, FreeMarker, Velocity, Thymeleaf, Struts тощо. Також, придбавши цю версію, з'являться інструменти для роботи з базами даних, SQL.

2.2 Модель програмного продукту

Коротко розглянемо структуру ПП, зображену на діаграмі класів.

1 Клас Tree23_Main– головний клас, оскільки містить в собі точку входу в програму. Також, в ньому створюється об'єкт типу класу Tree23 та містить меню, для керування виконання програми.

2 Клас Tree23 – клас, який відповідає, за всю функціональність ПП, оскільки в ньому реалізовані всі затребувані методи з ПЗ. Також, клас, тримає зв'язок з класом Node.

3 Клас Node – клас, який представляє собою зразок вузла дерева, з такими даними: лівий, центральний та правий наслідники; та зберігаємі, в цьому самому вузлі, елементи зліва та справа.

3 РЕАЛІЗАЦІЯ ПРИКЛАДНОГО РІШЕННЯ

3.1 Побудова програмного продукту

В результаті розробки програмного продукту (ПП), виконано все, що було заплановано та затребувалося прикладною задачею (ПЗ), критерії по реалізації ПП дотримані, характеристика усім компонентам надана, всі можливі результати і ситуації – враховані. Тому отримана структура ПП є коректною та складається з наступних класів, з переліками основних методів.

1 Клас `Tree23_Main` – головний клас, оскільки містить в собі точку входу в програму, у вигляді метода `main(String[] args)`. Також, в ньому створюється об'єкт типу класу `Tree23`. Окрім цього, він містить меню, з можливістю вибору всіх доступних операцій з 2-3 деревом, для більшого уявлення про клас, у додатку А представлений відповідний код реалізації.

2 Клас `Tree23` – клас, який відповідає за всю функціональність ПП, оскільки в ньому реалізовані всі затребувані методи з ПЗ, вихідний код представлено у додатку Б. Даний клас містить такі методи як:

а) `add(T element)` – метод, що зображений на рисунку 3.1, необхідний для вставка значення до дерева. Разом з цим, може відбуватися виклик методу для розділення, тобто, створення нової структури вузла за допомогою метода – `split(Node current, T element)`;

б) `split(Node current, T element)` – метод, який необхідний для правильного створення нової структури вузла;

в) `boolean remove(T element)` – метод, який необхідний для видалення вузла дерева, по передаваному значенню. Після процесу видалення відбувається перебалансування вузлів методом – `reBalance()`. Повертає логічне значення `true`, якщо видалення вдалося;

г) `T findMin()` – метод, для отримання мінімального значення в дереві;

д) `T findMax()` – метод, для отримання максимального значення в

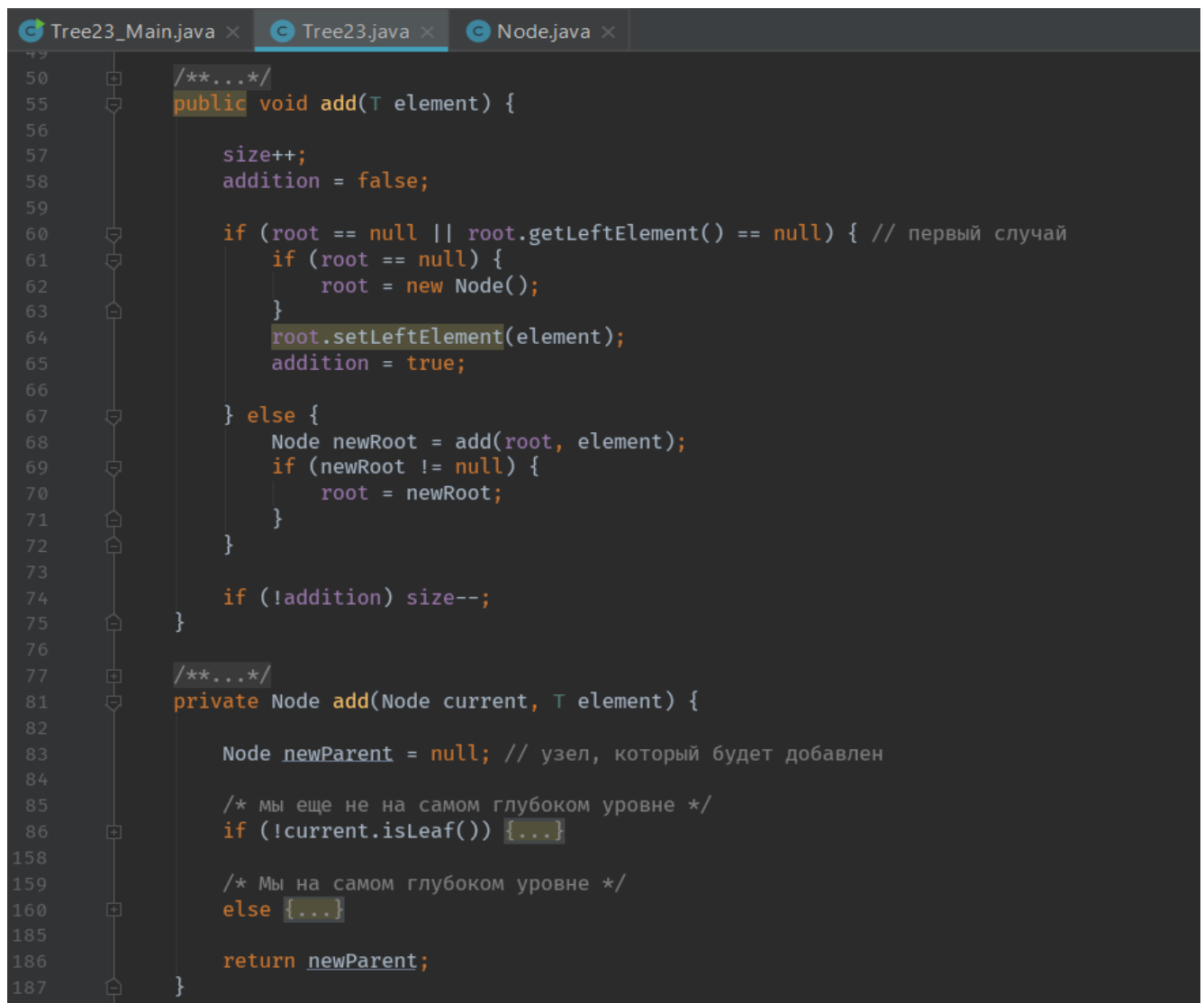
дереві;

е) `boolean search(T element)` – метод, за допомогою якого, можливий пошук відповідного значення, яке було передано з головного класу. Повертає логічне значення `true`, якщо пошук вдалий;

ж) `int size()` – метод, який підраховує розмір дерева;

з) `clear()` – метод, для очищення вмісту дерева;

и) `inOrder()`, `preOrder()`, `postOrder()` – методи, для друку значень дерева, різними способами (обходами).



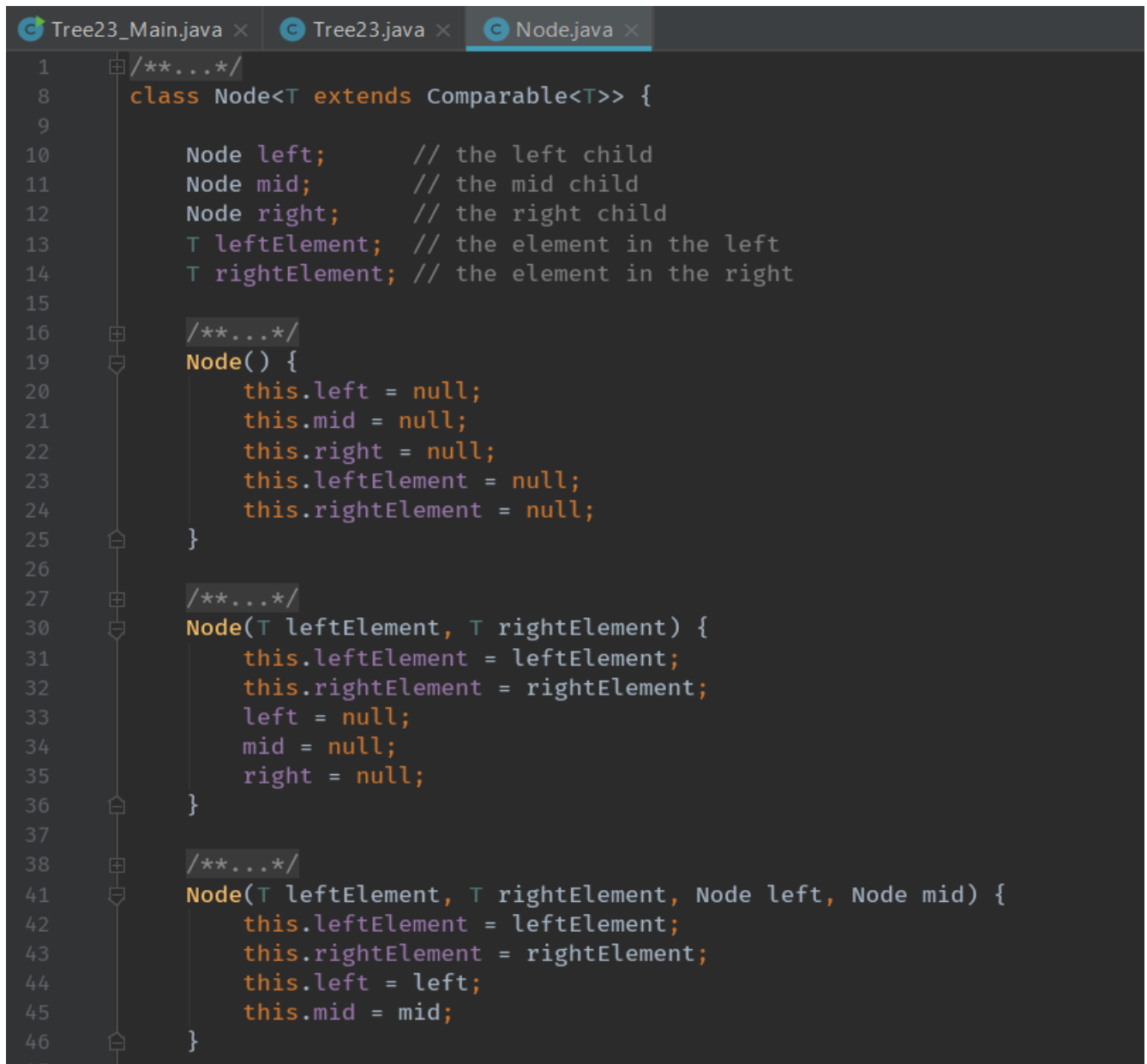
```

50  /**...*/
55  public void add(T element) {
56
57      size++;
58      addition = false;
59
60      if (root == null || root.getLeftElement() == null) { // первый случай
61          if (root == null) {
62              root = new Node();
63          }
64          root.setLeftElement(element);
65          addition = true;
66
67      } else {
68          Node newRoot = add(root, element);
69          if (newRoot != null) {
70              root = newRoot;
71          }
72      }
73
74      if (!addition) size--;
75  }
76
77  /**...*/
81  private Node add(Node current, T element) {
82
83      Node newParent = null; // узел, который будет добавлен
84
85      /* мы еще не на самом глубоком уровне */
86      if (!current.isLeaf()) {...}
158
159      /* Мы на самом глубоком уровне */
160      else {...}
185
186      return newParent;
187  }

```

Рисунок 3.1 – Вставка элемента до дерева

3 Клас Node – клас, який представляє собою зразок вузла дерева, з такими даними: лівий, центральний та правий наслідники; та зберігаємі, в цьому самому вузлі, елементи зліва та справа. Дані отримуються з класу Tree23 та обробляються (ініціалізуються значення об'єкту) конструктором. Вихідний код представлено у додатку В, а на рисунку 3.2, зображено поля та необхідні конструктори.



```

1  /**...*/
8  class Node<T extends Comparable<T>> {
9
10     Node left;        // the left child
11     Node mid;         // the mid child
12     Node right;       // the right child
13     T leftElement;    // the element in the left
14     T rightElement;   // the element in the right
15
16     /**...*/
19     Node() {
20         this.left = null;
21         this.mid = null;
22         this.right = null;
23         this.leftElement = null;
24         this.rightElement = null;
25     }
26
27     /**...*/
30     Node(T leftElement, T rightElement) {
31         this.leftElement = leftElement;
32         this.rightElement = rightElement;
33         left = null;
34         mid = null;
35         right = null;
36     }
37
38     /**...*/
41     Node(T leftElement, T rightElement, Node left, Node mid) {
42         this.leftElement = leftElement;
43         this.rightElement = rightElement;
44         this.left = left;
45         this.mid = mid;
46     }

```

Рисунок 3.2 – Вміст класу Node

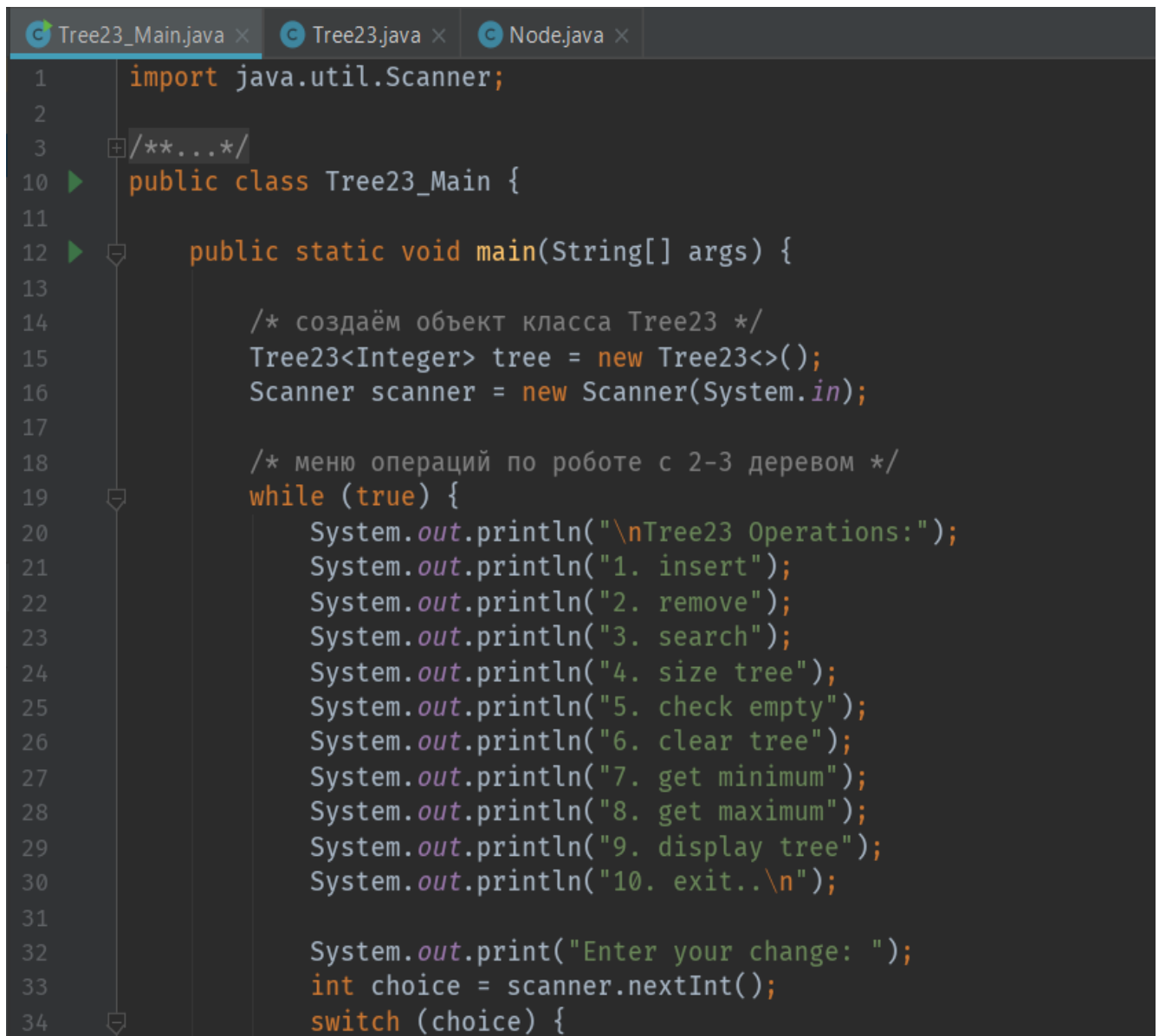
3.2 Тестування програмного продукту

Процес функціонування ПП, абстрактно, можна поділити на декілька стадій:

- 1) створення та ініціалізація відповідних полів;
- 2) отримання від користувача затребуваних пунктів меню, які представляють собою деякі завдання, та подальше їх оброблення в відповідних методах, з візуалізацією результатів на консолі IDE.

Протестуємо ПП на виявлення неточностей, помилок та багів, в кожній з виділених стадій окремо, за допомогою відлагоджувача програмного середовища IntelliJ IDEA.

Перша стадія представляє собою створення об'єкта класу Tree23. Одразу ж після цього, формується запит, у вигляді меню, на вибір необхідної процедури для модифікації або відображення 2-3 дерева, реалізація вище сказаного, зображена на рисунку 3.3.



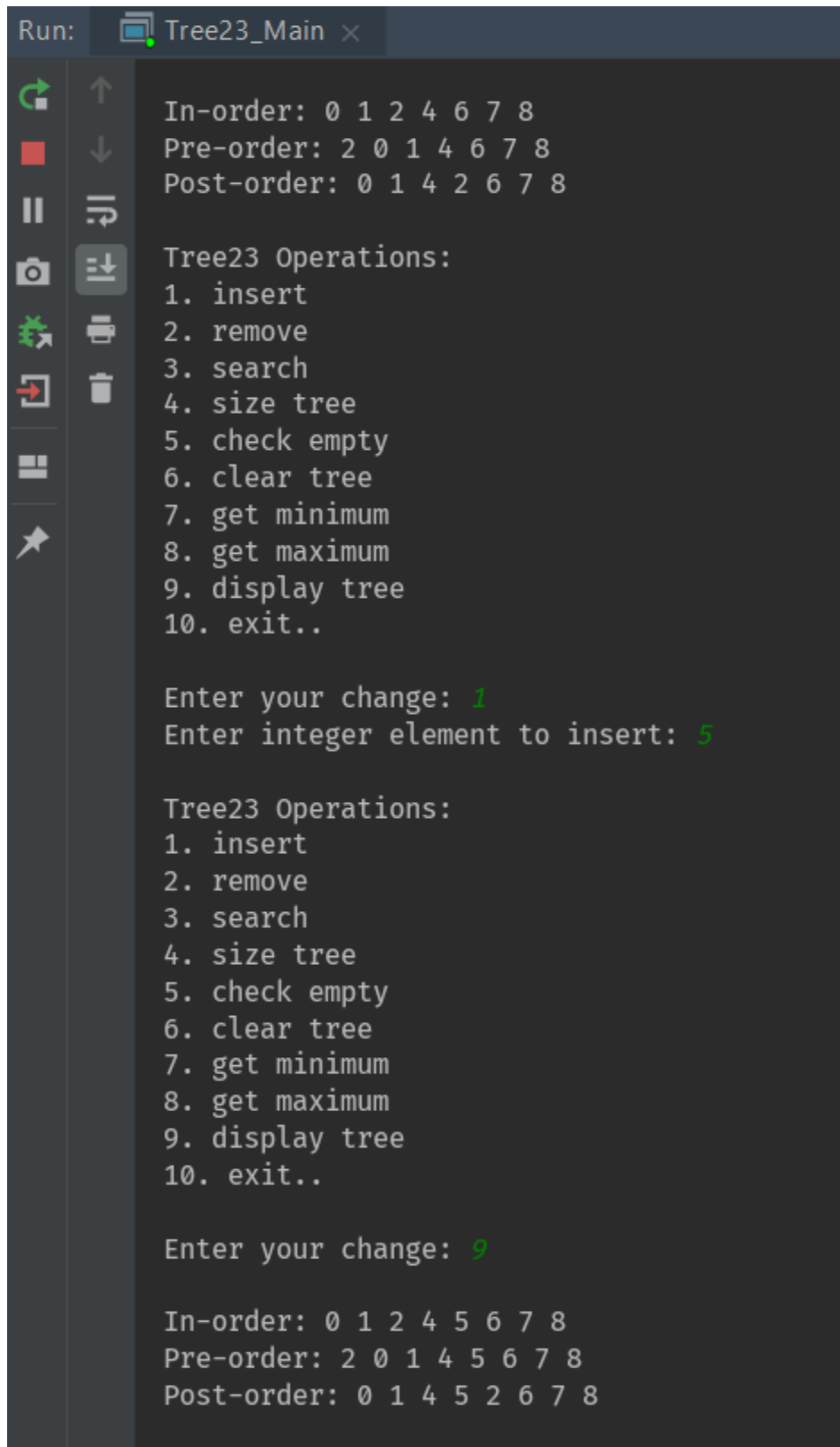
```

1  import java.util.Scanner;
2
3  /**...*/
10 public class Tree23_Main {
11
12     public static void main(String[] args) {
13
14         /* создаём объект класса Tree23 */
15         Tree23<Integer> tree = new Tree23<>();
16         Scanner scanner = new Scanner(System.in);
17
18         /* меню операций по работе с 2-3 деревом */
19         while (true) {
20             System.out.println("\nTree23 Operations:");
21             System.out.println("1. insert");
22             System.out.println("2. remove");
23             System.out.println("3. search");
24             System.out.println("4. size tree");
25             System.out.println("5. check empty");
26             System.out.println("6. clear tree");
27             System.out.println("7. get minimum");
28             System.out.println("8. get maximum");
29             System.out.println("9. display tree");
30             System.out.println("10. exit..\n");
31
32             System.out.print("Enter your change: ");
33             int choice = scanner.nextInt();
34             switch (choice) {

```

Рисунок 3.3 – Створення об'єкта 2-3 дерева та меню програми

Розроблена модель працює в автономному режимі, тому на другій стадії функціонування ПП, розпочнуть свою роботу основні процеси. До них належать всі методи з класу Tree23. Передбачена візуалізація результатів виконання роботи, як зображено на рисунку 3.4.



```
Run: Tree23_Main x
In-order: 0 1 2 4 6 7 8
Pre-order: 2 0 1 4 6 7 8
Post-order: 0 1 4 2 6 7 8

Tree23 Operations:
1. insert
2. remove
3. search
4. size tree
5. check empty
6. clear tree
7. get minimum
8. get maximum
9. display tree
10. exit..

Enter your change: 1
Enter integer element to insert: 5

Tree23 Operations:
1. insert
2. remove
3. search
4. size tree
5. check empty
6. clear tree
7. get minimum
8. get maximum
9. display tree
10. exit..

Enter your change: 9

In-order: 0 1 2 4 5 6 7 8
Pre-order: 2 0 1 4 5 6 7 8
Post-order: 0 1 4 5 2 6 7 8
```

Рисунок 3.4 – Візуалізація роботи методів вставки та друку

ВИСНОВОК

В роботі був описаний процес розробки програмної моделі (ПМ), яка представляє собою таку структури даних, як 2-3 дерево. ПМ працює, за певним сценарієм, тобто конфігурації вхідних даних (параметрів) та методів (дій), які будуть обробляти їх.

Була вивчена відповідна література по проблематиці реалізації такої структури даних, зокрема розглянуті ключові поняття та методики по роботі з нею. Сформоване розуміння методів, що дало змогу практично застосувати набуті знання. Тому, була розроблена відповідна демонстраційна програма.

Структура курсової роботи складається зі вступу, трьох розділів, висновку, списку джерел інформації та додатків.

В першому розділі, надані теоретичні відомості та область застосування прикладного рішення. Також, розглянуті переваги та недоліки алгоритму. Окрім цього, якісно описані основні методи, для роботи з 2-3 деревом, які представляють основний функціонал.

У другому розділі, детально розглянуті методи та модель прикладної задачі. Також, описані різноманітні інструментальні засоби, для реалізації та візуалізації програмного продукту.

В третьому розділі, були застосовані набуті знання, отримані з двох попередніх розділів. Тому, було детально описано реалізацію прикладного рішення, тобто, розглянуті основні положення програмного продукту та відповідно зображено його побудова та тестування.

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1 2-3-дерево [Електронний ресурс] – Режим доступу до ресурсу: <https://ru.wikipedia.org/wiki/2-3-дерево>.

2 2-3-дерево. Наивная реализация [Електронний ресурс] – Режим доступу до ресурсу: <https://habr.com/post/303374/>.

3 Б-деревя [Електронний ресурс] – Режим доступу до ресурсу: <http://saod.narod.ru/saod3/List013.html>.

4 Структура 2-3 дерева [Електронний ресурс] – Режим доступу до ресурсу: <https://studfiles.net/preview/4594013/page:8/#13>.

5 2-3 Trees [Електронний ресурс] – Режим доступу до ресурсу: <http://www.aihorizon.com/essays/basiccs/trees/twothree.htm>.

6 Kishori S. Beginning Java 8 Fundamentals. Language Syntax, Arrays, Data Types, Objects, and Regular Expressions. / Sharan Kishori., 2014. – 789 с. – (Apress).

7 Oracle. The Java Tutorials [Електронний ресурс] / Oracle – Режим доступу до ресурсу: <https://docs.oracle.com/javase/tutorial/java/index.html>.

8 Object Oriented Programming – Java OOPs Concepts With Examples [Електронний ресурс] – Режим доступу до ресурсу: <https://edureka.co/blog/object-oriented-programming>.

9 Fowler M. UML Distilled. A Brief Guide to the Standard Object Modeling Language / Martin Fowler. // Addison-Wesely. – 2004. – №3. – С. 175.

ДОДАТКИ

Додаток А. Лістинг класу Tree23_Main

```
import java.util.Scanner;

/**
 * Головний клас для роботи програми.
 * Содержить точку входу в програму.
 *
 * @author Syniuk Valentyn
 */
public class Tree23_Main {

    public static void main(String[] args) {

        /* Створюємо об'єкт класу Tree23 */
        Tree23<Integer> tree = new Tree23<>();
        Scanner scanner = new Scanner(System.in);

        /* Меню операцій по роботі з 2-3 деревом */
        while (true) {
            System.out.println("\nTree23 Operations:");
            System.out.println("1. insert");
            System.out.println("2. remove");
            System.out.println("3. search");
            System.out.println("4. size tree");
            System.out.println("5. check empty");
            System.out.println("6. clear tree");
            System.out.println("7. get minimum");
            System.out.println("8. get maximum");
            System.out.println("9. display tree");
            System.out.println("10. exit..\n");

            System.out.print("Enter your change: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1: {
                    System.out.print("Enter integer element to
insert: ");

                    tree.add(scanner.nextInt());
                    break;
                }
                case 2: {
                    System.out.print("Enter integer element to
remove: ");
```


Додаток Б. Лістинг класу Tree23

```

/**
 * Класс реализует структуру данных "2-3 дерево".
 * Такая структура хранит элементы в виде древовидной структуры, но
 * сбалансированные.
 *
 * @author Syniuk Valentyn
 */
class Tree23<T extends Comparable<T>> {

    private static final int ROOT_IS_BIGGER = 1;
    private static final int ROOT_IS_SMALLER = -1;

    private Node root;           // Корень дерева
    private int size;            // Количество элементов дерева
    private boolean addition;    // Флаг, чтобы знать, был ли последний
    элемент добавлен правильно или нет

    /**
     * Конструктор класса
     */
    Tree23() {
        this.root = new Node();
        this.size = 0;
    }

    /**
     * @return true, если дерево пусто, иначе false
     */
    public boolean isEmpty() {
        if (root == null) return true;
        return root.getLeftElement() == null;
    }

    /**
     * Метод, для проверки, содержится ли данный узел в дереве
     *
     * @param element элемент для поиска
     * @return true, если это дерево содержит указанный элемент,
    иначе false
     */
    public boolean contains(T element) {
        return search(element);
    }

    /**
     * @return кол-во элементов дерева

```

```

    */
    public int size() {
        return size;
    }

    /**
     * Метод, для добавления нового элемента в дерево, сохраняя его
     * сбалансированным
     *
     * @param element элемент для добавления
     */
    public void add(T element) {

        size++;
        addition = false;

        if (root == null || root.getLeftElement() == null) { //
первый случай
            if (root == null) {
                root = new Node();
            }
            root.setLeftElement(element);
            addition = true;

        } else {
            Node newRoot = add(root, element);
            if (newRoot != null) {
                root = newRoot;
            }
        }

        if (!addition) size--;
    }

    /**
     * @param current узел, в который нужно добавить
     * @param element элемент для вставки
     */
    private Node add(Node current, T element) {

        Node newParent = null; // узел, который будет добавлен

        /* мы еще не на самом глубоком уровне */
        if (!current.isLeaf()) {

            Node newNode;

            /* такой элемент уже существует */

```

```

        if (current.leftElement.compareTo(element) == 0 ||
(current.is3Node() && current.rightElement.compareTo(element) == 0))
{
    }

    // newNode меньше левого элемента
    else if (current.leftElement.compareTo(element) ==
ROOT_IS_BIGGER) {
        newNode = add(current.left, element);

        // newNode приходит из левой ветви
        if (newNode != null) {

            // newNode, в этом случае, всегда меньше, чем
current.left

            if (current.is2Node()) {
                current.rightElement = current.leftElement;
// сдвинуть текущий левый элемент вправо
                current.leftElement = newNode.leftElement;
                current.right = current.mid;
                current.mid = newNode.mid;
                current.left = newNode.left;
            }

            // В этом случае у нас новое разделение, поэтому
текущий элемент слева будет подниматься
            else {

                // копируем правую часть поддерева
                Node rightCopy = new
Node(current.rightElement, null, current.mid, current.right);

                // создаем новую «структуру», вставляя правую
часть
                newParent = new Node(current.leftElement,
null, newNode, rightCopy);
            }
        }

        // newNode больше левого и меньше правого
        else if (current.is2Node() || (current.is3Node() &&
current.rightElement.compareTo(element) == ROOT_IS_BIGGER)) {

            newNode = add(current.mid, element);

            // новое разделение
            if (newNode != null) {

```

```

        // правый элемент пуст, поэтому мы можем
установить newNode слева, а существующий левый элемент - справа
        if (current.is2Node()) {
            current.rightElement = newNode.leftElement;
            current.right = newNode.mid;
            current.mid = newNode.left;
        }

        // еще один случай, когда мы должны снова
разделить
        else {
            Node left = new Node(current.leftElement,
null, current.left, newNode.left);
            Node mid = new Node(current.rightElement,
null, newNode.mid, current.right);
            newParent = new Node(newNode.leftElement,
null, left, mid);
        }
    }

    // newNode больше, чем правый элемент
    else if (current.is3Node() &&
current.rightElement.compareTo(element) == ROOT_IS_SMALLER) {

        newNode = add(current.right, element);

        // разделение -> правый элемент поднимается
        if (newNode != null) {
            Node leftCopy = new Node(current.leftElement,
null, current.left, current.mid);
            newParent = new Node(current.rightElement, null,
leftCopy, newNode);
        }
    }

    /* Мы на самом глубоком уровне */
    else {

        addition = true;

        /* такой элемент уже существует */
        if (current.leftElement.compareTo(element) == 0 ||
(current.is3Node() && current.rightElement.compareTo(element) == 0))
        {
            addition = false;
        }
    }

```



```

        /* случай, когда нет правильного элемента */
        else if (current.is2Node()) {

            // если текущий левый элемент больше чем newNode, мы
            сдвигаем левый элемент вправо
            if (current.leftElement.compareTo(element) ==
ROOT_IS_BIGGER) {
                current.rightElement = current.leftElement;
                current.leftElement = element;
            }

            // если newNode больше, мы добавляем его справа
            else if (current.leftElement.compareTo(element) ==
ROOT_IS_SMALLER) current.rightElement = element;
        }

        /* Случай когда в узле 2 элемента, и мы хотим добавить
        еще один. Для этого мы разделяем узел */
        else newParent = split(current, element);
    }

    return newParent;
}

/**
 * Создает новую структуру узла, которая будет присоединена в
нижней части метода add
 *
 * @param current узел, где происходит разделение
 * @param element элемент для вставки
 * @return двухузловая структура с ненулевым левым и средним
узлом
 */
private Node split(Node current, T element) {

    Node newParent = null;

    /* левый элемент больше, поэтому он будет подниматься,
    позволяя встать newParent слева */
    if (current.leftElement.compareTo(element) == ROOT_IS_BIGGER)
    {
        Node<T> left = new Node<>(element, null);
        Node right = new Node(current.rightElement, null);
        newParent = new Node(current.leftElement, null, left,
right);
    } else if (current.leftElement.compareTo(element) ==
ROOT_IS_SMALLER) {

```

```

        // newParent больше текущего справа и меньше правого
        элемента
        // newParent поднимается
        if (current.rightElement.compareTo(element) ==
ROOT_IS_BIGGER) {

            Node left = new Node(current.leftElement, null);
            Node right = new Node(current.rightElement, null);
            newParent = new Node(element, null, left, right);

        }

        // newParent самый большой, поэтому текущий правый
        элемент поднимается
        else {

            Node left = new Node(current.leftElement, null);
            Node<T> right = new Node<>(element, null);
            newParent = new Node(current.rightElement, null,
left, right);
        }

        return newParent;
    }

    /**
     * Метод, для удаления элемента из дерева
     *
     * @param element элемента для удаления
     * @return true, если элемент был удалён, иначе false
     */
    public boolean remove(T element) {
        boolean ifRemoved;

        // уменьшаем кол-во уровней в начале
        this.size--;

        ifRemoved = remove(root, element);

        root.reBalance();

        // если удалили последний элемент дерева
        if (root.getLeftElement() == null) root = null;

        // если элемент не был удалён, увеличиваем кол-во уровней
        if (!ifRemoved) this.size++;

        return ifRemoved;
    }

```

```

}

/**
 * @param current узел из которого нужно удалить
 * @param element элемент который нужно удалить
 * @return true, если элемент был удалён, иначе false
 */
private boolean remove(Node current, T element) {
    boolean ifRemoved = true;

    /* Случай, когда мы находимся в самом глубоком уровне дерева,
но мы не нашли элемент (он не существует) */
    if (current == null) {
        ifRemoved = false;
        return false;
    }

    /* Рекурсивный случай, мы все еще находим элемент для
удаления */
    else {

        if (!current.getLeftElement().equals(element)) {

            // Если справа нет элемента или удаляемый элемент
меньше правого элемента
            if (current.getRightElement() == null ||
current.getRightElement().compareTo(element) == ROOT_IS_BIGGER) {

                // Левый элемент больше, чем удаляемый элемент,
поэтому мы проходим левый дочерний элемент
                if (current.getLeftElement().compareTo(element)
== ROOT_IS_BIGGER) {
                    ifRemoved = remove(current.left, element);
                }

                // Иначе проходим средний дочерний элемент
                else {
                    ifRemoved = remove(current.mid, element);
                }

            } else {

                // Если удаляемый элемент не равен нужному
элементу, мы проходим правого потомка
                if (!current.getRightElement().equals(element)) {
                    ifRemoved = remove(current.right, element);
                }

                // Иначе, мы нашли элемент

```

```

else {

    // *** Ситуация 1 ***
    // Элемент равен правому элементу листа,
    поэтому мы просто удаляем его
    if (current.isLeaf()) {
        current.setRightElement(null);
    }

    // *** Ситуация 2 ***
    // Мы нашли элемент, но его нет в листе
    else {

        // Мы получаем элемент "min" правой
        ветви, удаляем его из текущей позиции и помещаем туда,
        // где нашли элемент для удаления
        T replacement = (T)
        current.getRightNode().replaceMin();

        current.setRightElement(replacement);
    }
}

/* Левый элемент равен элементу для удаления */
else {

    // *** Ситуация 1 ***
    if (current.isLeaf()) {

        // Левый элемент, элемент для удаления,
        заменяется правым элементом
        if (current.getRightElement() != null) {

            current.setLeftElement(current.getRightElement());
            current.setRightElement(null);

        }

        // Если справа нет элемента, потребуется
        перебалансировка

        else {
            current.setLeftElement(null); // Отпускаем
            узел

            return true;
        }
    }
}

```

```

// *** Ситуация 2 ***
else {

    // Перемещаем элемент "max" левой ветви, где мы
нашли элемент
    T replacement = (T)
current.getLeftNode().replaceMax();
    current.setLeftElement(replacement);
}
}

// Нижний уровень должен быть сбалансирован
if (current != null && !current.isBalanced()) {
    current.reBalance();
} else if (current != null && !current.isLeaf()) {

    boolean isBalanced = false;

    while (!isBalanced) {

        if (current.getRightNode() == null) {

            // Критический случай ситуации 2 у левого потомка
            if (current.getLeftNode().isLeaf() &&
!current.getMidNode().isLeaf()) {
                T replacement = (T)
current.getMidNode().replaceMin();
                T tempLeft = (T) current.getLeftElement();
                current.setLeftElement(replacement);

                add(tempLeft);
            }

            // Критический случай ситуации 2 у правого
потомка
            else if (!current.getLeftNode().isLeaf() &&
current.getMidNode().isLeaf()) {

                if (current.getRightElement() == null) {
                    T replacement = (T)
current.getLeftNode().replaceMax();
                    T tempLeft = (T)
current.getLeftElement();
                    current.setLeftElement(replacement);

                    add(tempLeft);
                }
            }
        }
    }
}

```

```

    }

    if (current.getRightNode() != null) {

        if (current.getMidNode().isLeaf() &&
!current.getRightNode().isLeaf()) {
            current.getRightNode().reBalance();
        }

        if (current.getMidNode().isLeaf() &&
!current.getRightNode().isLeaf()) {
            T replacement = (T)
current.getRightNode().replaceMin();
            T tempRight = (T) current.getRightElement();
            current.setRightElement(replacement);

            add(tempRight);
        } else {
            isBalanced = true;
        }
    }

    if (current.isBalanced()) isBalanced = true;
}

return ifRemoved;
}

/**
 * Метод, для удаления всех элементов из дерева
 */
public void clear() {
    this.size = 0;
    this.root = null;
}

/**
 * Метод, для поиска элемента в дереве
 *
 * @param element элемент, который нужно найти
 * @return true, если элемент был найден, иначе false
 */
public boolean search(T element) {
    if (root == null) return false;
    return search(root, element);
}

private boolean search(Node current, T element) {

```

```

    boolean ifFound = false;

    if (current != null) {

        /* В тривиальном случае -> мы нашли элемент */
        if (current.leftElement != null &&
current.leftElement.equals(element)) {
            ifFound = true;
        }

        /* Мы еще не на самом глубоком уровне */
        else {

            // Элемент для поиска равен правому элементу
            if (current.rightElement != null &&
current.rightElement.equals(element)) {
                ifFound = true;
            }

            // иначе -> рекурсивные вызовы
            else {
                if (current.leftElement.compareTo(element) ==
ROOT_IS_BIGGER) {
                    ifFound = search(current.left, element);

                } else if (current.right == null ||
current.rightElement.compareTo(element) == ROOT_IS_BIGGER) {
                    ifFound = search(current.mid, element);

                } else if
(current.rightElement.compareTo(element) == ROOT_IS_SMALLER) {
                    ifFound = search(current.right, element);

                } else return false;
            }
        }

        return ifFound;
    }

/**
 * Метод, для поиска минимального значения
 *
 * @return найденное минимальное значение, иначе null
 */
public T findMin() {
    if (isEmpty()) return null;
    return findMin(root);
}

```

```

    }

    private T findMin(Node current) {

        // получаем минимальный элемент
        if (current.getLeftNode() == null) {
            return (T) current.leftElement;
        }

        // иначе -> рекурсивные вызовы
        else {
            return findMin(current.getLeftNode());
        }
    }

    /**
     * Метод, для поиска максимального значения
     *
     * @return найденное максимальное значение, иначе null
     */
    public T findMax() {
        if (isEmpty()) return null;
        return findMax(root);
    }

    private T findMax(Node current) {

        // рекурсивные вызовы
        if (current.rightElement != null && current.getRightNode() !=
null) {
            return findMax(current.getRightNode());
        } else if (current.getMidNode() != null) {
            return findMax(current.getMidNode());
        }

        // получаем максимальный элемент
        if (current.rightElement != null) return (T)
current.rightElement;
        else return (T) current.leftElement;
    }

    /**
     * Метод, для вывода элементов дерева в порядке способа - "in-
order"
     */
    public void inOrder() {

        if (!isEmpty()) inOrder(root);
        else System.out.print("The tree is empty...");
    }

```



```

    }

    private void inOrder(Node current) {

        if (current != null) {

            if (current.isLeaf()) {

                System.out.print(current.getLeftElement().toString()
+ " ");

                if (current.getRightElement() != null) {

System.out.print(current.getRightElement().toString() + " ");

                }
            } else {

                inOrder(current.getLeftNode());
                System.out.print(current.getLeftElement().toString()
+ " ");

                inOrder(current.getMidNode());

                if (current.getRightElement() != null) {

                    if (!current.isLeaf()) {

System.out.print(current.getRightElement().toString() + " ");

                    }

                    inOrder(current.getRightNode());

                }

            }

        }

    }

    /**
     * Метод, для вывода элементов дерева в порядке способа - "pre-
order"
     */
    public void preOrder() {

        if (!isEmpty()) {

            preOrder(root);
        } else System.out.print("The tree is empty...");

    }

    private void preOrder(Node current) {

        if (current != null) {

```

```

        System.out.print(current.leftElement.toString() + " ");
        preOrder(current.left);
        preOrder(current.mid);

        if (current.rightElement != null) {

            System.out.print(current.rightElement.toString() + "
");
            preOrder(current.right);
        }
    }

    /**
     * Метод, для вывода элементов дерева в порядке способа - "post-
    order"
     */
    public void postOrder() {

        if (!isEmpty()) {

            postOrder(root);
        } else System.out.print("The tree is empty...");
    }

    private void postOrder(Node current) {

        if (current != null) {

            postOrder(current.left);
            postOrder(current.mid);
            System.out.print(current.leftElement.toString() + " ");

            if (current.rightElement != null) {

                System.out.print(current.rightElement.toString() + "
");
                postOrder(current.right);
            }
        }
    }
}

```

Додаток В. Лістинг класу Node

```

/**
 * Дерево 2-3 образовано узлами, в которых хранятся элементы
структуры.
 * Каждый узел содержит не более двух элементов и не менее одного.
 *
 * @author Syniuk Valentyn
 */
class Node<T extends Comparable<T>> {

    Node left;        // the left child
    Node mid;         // the mid child
    Node right;       // the right child
    T leftElement;    // the element in the left
    T rightElement;   // the element in the right

    /**
     * Конструктор. Создает пустой node/child
     */
    Node() {
        this.left = null;
        this.mid = null;
        this.right = null;
        this.leftElement = null;
        this.rightElement = null;
    }

    /**
     * Конструктор 3-х узлов без определенных потомков (нулевые
ссылки).
     */
    Node(T leftElement, T rightElement) {
        this.leftElement = leftElement;
        this.rightElement = rightElement;
        left = null;
        mid = null;
        right = null;
    }

    /**
     * Конструктор 3-х узлов с заданными левым и средним
узлами/потомками.
     */
    Node(T leftElement, T rightElement, Node left, Node mid) {
        this.leftElement = leftElement;
        this.rightElement = rightElement;
        this.left = left;

```

```

        this.mid = mid;
    }

    T getLeftElement() {
        return leftElement;
    }

    void setLeftElement(T element) {
        this.leftElement = element;
    }

    T getRightElement() {
        return rightElement;
    }

    void setRightElement(T element) {
        this.rightElement = element;
    }

    private void setLeftNode(Node left) {
        this.left = left;
    }

    Node getLeftNode() {
        return left;
    }

    private void setMidNode(Node mid) {
        this.mid = mid;
    }

    Node getMidNode() {
        return mid;
    }

    private void setRightNode(Node right) {
        this.right = right;
    }

    Node getRightNode() {
        return right;
    }

    /**
     * @return true, если мы находимся на самом глубоком уровне
     * дерева, иначе false
     */
    boolean isLeaf() {
        return left == null && mid == null && right == null;
    }

```

```

}

/**
 * @return true, если правого узла не существует, иначе false
 */
boolean is2Node() {
    return rightElement == null;
}

/**
 * @return true, если правый узел существует, иначе false
 */
boolean is3Node() {
    return rightElement != null;
}

/**
 * Метод, для проверки, хорошо ли сбалансировано дерево
 *
 * @return true если дерево хорошо сбалансировано, иначе false
 */
boolean isBalanced() {

    boolean balanced = false;

    if (isLeaf()) { // Если мы находимся на самом глубоком уровне
(лист), он точно сбалансирован
        balanced = true;

    } else if (left.getLeftElement() != null &&
mid.getLeftElement() != null) { // Есть два случая: 2 узла или 3 узла

        if (rightElement != null) { // 3 узла
            if (right.getLeftElement() != null) {
                balanced = true;
            }
        } else { // 2 узла
            balanced = true;
        }
    }

    return balanced;
}

/**
 * @return максимальный элемент
 */
T replaceMax() {

```

```

    T max;

    /* Тривиальный случай, мы находимся на самом глубоком уровне
дерева */
    if (isLeaf()) {

        if (getRightElement() != null) {
            max = getRightElement();
            setRightElement(null);
            // Нам повезло, нам ничего не нужно перебалансировать
        } else {
            max = getLeftElement();
            setLeftElement(null);
            // На первом этапе рекурсивной функции произойдет
перебалансировка
        }
    }

    /* Рекурсивный случай, мы не на самом глубоком уровне */
    else {

        // Если есть элемент справа, мы продолжаем справа
        if (getRightElement() != null) {
            max = (T) right.replaceMax();
        }

        // Иначе, продолжаем со средним
        else {
            max = (T) mid.replaceMax();
        }
    }

    /* Сохраняем баланс */
    if (!isBalanced()) {
        reBalance();
    }

    return max;
}

/**
 * @return минимальный элемент
 */
T replaceMin() {

    T min;

    /* Тривиальный случай, мы находимся на самом глубоком уровне
дерева */

```

```

    if (isLeaf()) {

        min = leftElement;
        leftElement = null;

        // Элемент был справа, мы пропустили его слева, и здесь
        // ничего не произошло
        if (rightElement != null) {
            leftElement = rightElement;
            rightElement = null;
        }
    }

    /* Рекурсивный случай, пока мы не достигаем самого глубокого
    уровня, мы всегда спускаемся влево */
    else {
        min = (T) left.replaceMin();
    }

    // Сохраняем баланс
    if (!isBalanced()) {
        reBalance();
    }

    return min;
}

/**
 * Метод, для сохранения баланса, путём перебалансировки
 * самого глубокого уровня дерева начиная со второго самого
    глубокого
 */
void reBalance() {

    while (!isBalanced()) {

        /* Дисбаланс в левом потомке */
        if (getLeftNode().getLeftElement() == null) {

            // Мы ставим левый элемент текущего узла как левый
            // элемент левого потомка
            getLeftNode().setLeftElement(getLeftElement());

            // Теперь мы заменяем левый элемент среднего потомка
            // как левый элемент текущего узла
            setLeftElement((T) getMidNode().getLeftElement());

            // Если правый элемент на среднем дочернем элементе
            // существует, мы сдвигаем его влево

```

```

        if (getMidNode().getRightElement() != null) {
getMidNode().setLeftElement(getMidNode().getRightElement());
        getMidNode().setRightElement(null);
        }

        // Иначе, мы дадим среднему потомку "empty", так что
        // следующая итерация может разрешить эту ситуацию,
        // если нет, то здесь начинается критический случай
        else {
            getMidNode().setLeftElement(null);
        }
    }

    /* Дисбаланс в правом ребенке */
    else if (getMidNode().getLeftElement() == null) {

        // Критический случай, каждый узел (дочерний элемент)
        // самого глубокого уровня
        // имеет только один элемент, алгоритм должен будет
        // выполнить балансировку с более высокого уровня дерева
        if (getRightElement() == null) {

            if (getLeftNode().getLeftElement() != null &&
getLeftNode().getRightElement() == null &&
getMidNode().getLeftElement() == null) {
                setRightElement(getLeftElement());
                setLeftElement((T)
getLeftNode().getLeftElement());

                // мы удаляем текущих потомков
                setLeftNode(null);
                setMidNode(null);
                setRightNode(null);

            } else {

getMidNode().setLeftElement(getLeftElement());
                if (getLeftNode().getRightElement() == null)
                {
                    setLeftElement((T)
getLeftNode().getLeftElement());
                    getLeftNode().setLeftElement(null);

                } else {
                    setLeftElement((T)
getLeftNode().getRightElement());
                    getLeftNode().setRightElement(null);
                }
            }
        }
    }
}

```



```

    }

    if (getLeftNode().getLeftElement() == null &&
getMidNode().getLeftElement() == null) {
        setLeftNode(null);
        setMidNode(null);
        setRightNode(null);
    }
}

} else {

    // Мы ставим правый элемент текущего узла как
    левый элемент среднего сына
    getMidNode().setLeftElement(getRightElement());

    // Мы помещаем левый элемент правого потомка в
    качестве правого элемента текущего узла
    setRightElement((T)
getRightNode().getLeftElement());

    // Если правый дочерний элемент, в котором мы
    взяли последний элемент,
    // имеет правый элемент, мы перемещаем его слева
    от того же дочернего элемента.
    if (getRightNode().getRightElement() != null) {
getRightNode().setLeftElement(getRightNode().getRightElement());
        getRightNode().setRightElement(null);

    }

    // Иначе мы дадим нужному ребенку "empty"
    else {
        getRightNode().setLeftElement(null);
    }
}

}

/* Дисбаланс в правом потомке */
else if (getRightElement() != null &&
getRightNode().getLeftElement() == null) {

    // *** Ситуация 1 ***
    // Средний ребенок существует, поэтому мы делаем
    сдвиг элементов вправо
    if (getMidNode().getRightElement() != null) {
        getRightNode().setLeftElement(getRightElement());
    }
}

```

```

        setRightElement((T)
getMidNode().getRightElement());
        getMidNode().setRightElement(null);

    }

    // *** Ситуация 2 ***
    // Средний дочерний элемент имеет только левый
элемент, тогда мы должны поместить правый элемент текущего узла в
качестве правого элемента среднего дочернего элемента.
    else {
        getMidNode().setRightElement(getRightElement());
        setRightElement(null);
    }
}
}
}
}
}

```